Dipanshu Gandhi 04/06/2023

GB3: Final Report

1. Introduction

RISC-V processors are built upon the RISC-V instruction set, which is a contemporary, open-source architecture aimed at fostering open innovation and facilitating the development of open-source CPU designs. The objective of this project was to optimize the design of a RISC-V processor, with a focus on enhancing performance, reducing power consumption, and minimizing resource utilization, ultimately leading to a Pareto optimal design. To achieve the Pareto optimal design, we proposed improvements to the CPU's architecture after conducting a thorough analysis of its design and measuring its runtime performance. The motivation behind pursuing a Pareto-optimal design is that prioritizing one characteristic of the CPU at the expense of others would result in an inadequate general-purpose CPU. For instance, favouring performance improvements might necessitate a larger number of components, leading to increased costs, resource usage, and power consumption. In this project, soft-core CPU designs are executed directly on the FPGA, enabling the incorporation of the FPGA's existing built-in logic and function blocks into the CPU's design. For instance, we utilized the FPGA's DSP to decrease the ALU (arithmetic logic unit) runtime, resulting in improved performance (refer to sections 2 and 3).

To begin, we implemented a RISC-V processor on a Lattice ICE40 UltraPlus FPGA board and utilized it as a soft-core. This approach allowed for rapid design modifications and testing. Furthermore, it facilitated I/O interfacing with the CPU, enabling measurement of the CPU program runtime (detailed in section 4 - test procedure). To approximate the number of instructions executed by each program, we employed the Sunflower CPU simulation suite. Combining these parameters, as explained in section 4, we determined program runtime, total number of instructions executed, clock rate, and cycles per instruction (CPI) – a crucial metric we aimed to improve. Additionally, custom test programs were written in C to specifically emphasize various aspects of the CPU, such as the branch predictor. These specialized tests aided in better analysing the benefits of the modifications (further details provided in section 4). Within our team, I focused on enhancing the CPU's performance. CPI improvements were pursued through architectural enhancements, particularly targeting pipeline stalls and hazards, while the clock rate was enhanced by reducing the critical path. The design process involved an iterative approach, with each improvement being tested by analysing the output of synthesis and place-and-route tools, as well as evaluating the performance of the modified CPU designs when executed on the FPGA. In addition, the aforementioned custom C tests were used to verify the performance and correctness of the modified design. For example, when the ALU was replaced with a DSP block, the correctness of the implementation was checked to ensure that the DSP provided the same numerical results as the original ALU block. Overall, our group was able to achieve the following results (see Fig 1.1), where it can be observed that a trade-off was made for increasing performance (with an increase in CPI and decrease in runtime) at the expense of increased resource usage. These results were found using bubblesort (with minor modifications), as it is a comprehensive test of all of the CPU's components. Further results are included in section 4.

Attribute	Original	Improved design
Resource usage (logic cells)	3128	3961
Bubblesort runtime (seconds)	4.192	1.994
Clock frequency	6 MHz	12 MHz
CPI	1.21	1.14
Power consumption (mA)	Not meaningfully different across any of the 6 measurement points	

Figure 1.1: Comparison of original and final improved design

2. Design Strategy

Multiple strategies were employed to enhance the processor's performance, with a primary focus on augmenting the CPU's clock cycle. The clock cycle is crucial as it synchronizes the sequential components of the CPU, enabling them to respond at the positive edge of the clock signal. By increasing the clock cycle, the processor's runtime is directly reduced. This inverse correlation between the program's runtime and the CPU's clock frequency suggests that elevating the clock frequency decreases the overall cycle count required for program execution. For example, if the CPU requires 10 cycles to execute a single instruction and a program consists of 1000 instructions, the program will be completed in 10,000 cycles. This example clearly illustrates the advantages derived from higher CPU clock frequencies (cycles per second). However, it is essential to acknowledge that an increased clock rate can potentially amplify power consumption in the CPU. Power usage is predominantly attributed to the switching of CMOS logic within the FPGA. Consequently, a higher clock rate results in more frequent logic switches per unit time, leading to escalated power consumption. Nevertheless, when the program's runtime is significantly shorter, the power draw occurs over a shorter duration. Thus, striking a delicate equilibrium between performance and power efficiency depends on the intended purpose of the CPU. Despite these considerations, prioritizing performance improvement is favoured since the CPU can be deactivated during periods of inactivity. It is worth noting that employing clock gating in a more advanced design could mitigate static power consumption. However, due to time and subject knowledge limitations, this approach was not pursued in this project.

To enhance the CPU's clock cycle, efforts were made to reduce the critical path length, which represents the longest combinational path between clock cycles. The critical path includes a critical propagation delay that determines the minimum clock period and limits the maximum clock frequency. Examination of the bubble sort algorithm on the original processor design unveiled a critical path consisting of 50 logic levels and a path delay of 74.30 nanoseconds, as depicted in Figure 6.1 in the appendix. This path traverses the forwarding unit and encounters delays caused by carry signal propagation in the ALU, as indicated by the hierarchical signal net names in the terminal output of the place-and-route tool. Multiple strategies were employed to diminish the critical delay. Initially, increasing the pipeline depth by synchronising the multiplexers and forwarding units was attempted. While this approach extends the clock cycles required for each instruction to complete, it raises the potential for clock cycle improvement. Section 3 will conduct calculations to determine the maximum viable pipeline depth and evaluate whether it results in enhanced performance despite a lower CPI (cycles per instruction). Additionally, the ALU's addition and subtraction operations were implemented using the FPGA's DSP blocks, as these blocks are optimized for rapid execution of these operations.

Furthermore, attention was given to improving the CPU's branch prediction mechanisms. This is significant as it positively impacts the processor's CPI (cycles per instruction), enabling faster program completion without substantial increases in power consumption and bringing us closer to a Pareto-optimal design. Control hazards in the CPU's pipeline occur when incorrect predictions are made regarding preceding branch instructions, leading to unnecessary loading of instructions. Consequently, the CPU experiences stalls, pipeline flushing, and restarting from a point prior to the hazard. This can result in significant increases in program runtime, as illustrated in Figure 6.2 in the appendix. By designing a more effective branch predictor, we anticipate a higher accuracy in predicting the outcomes of branching instructions, leading to fewer pipeline stalls. Various types of branch predictors exist, but the focus was placed on implementing local, global, and tournament branch history predictors. Given more time, a more advanced algorithm like gselect/gshare (which combines local and global history predictors optimally) could have been implemented. However, such an implementation would have significantly increased

resource usage (as outlined in section 3) and proved complex to accomplish within the limited timeframe. Sections 3 and 4 provide more detailed information on the chosen branch predictor and rationale behind the specific configuration employed.

3. Design Description & Problems Encountered

In this section, I will outline the designs that I attempted to implement, outlining any problems that I faced.

3.1 Implementing DSP blocks

In the pursuit of enhancing the performance of the Arithmetic Logic Unit (ALU), the initial exploration involved investigating the replacement of the addition and subtraction circuitry with Field-Programmable Gate Arrays (FPGA) Digital Signal Processing (DSP) blocks. Two approaches were considered: explicit instantiation of FPGA blocks by defining them in the Verilog code, or allowing the synthesis program to automatically infer the suitable blocks. However, the initial synthesis results revealed an underutilization of the FPGA's DSP blocks, as depicted in Figure 6.3 in the appendix. To address this, explicit definition of the DSP blocks as 32-bit adder/subtractors within the ALU's Verilog code was implemented, as shown in Figure 6.4 in the appendix. Since the DSPs operate on a clocked component, the global clock signal needed to be passed to them, even if the inputs and outputs were not registered. Wiring the 10+ input signals of the DSP blocks was necessary to enable their implementation. To streamline the DSP implementation, the utilization of a separate module to handle the switch between addition and subtraction was foregone. While this approach may not have been optimal in terms of resource usage, it could be addressed in subsequent iterations of the CPU design. However, due to the project's time constraints, the utilization of a single DSP was avoided. Furthermore, investigation was conducted to explore the potential optimization of other ALU operations using DSP blocks, particularly the branch-enable operations involving number comparisons, which were identified as the critical path, as confirmed by the output terminal in Figure 6.5 in the appendix.

One potential improvement considered was performing comparisons between signed numbers using DSP subtraction, which is highly efficient, and subsequently evaluating the Most Significant Bit (MSB) to determine the order of the numbers, thereby ascertaining whether the result was positive or negative. Unfortunately, due to time limitations, the implementation of this approach could not be accomplished. In retrospect, it would have been a relatively straightforward enhancement, and had the unsuccessful attempts been foreseen, prioritization of this approach would have been considered earlier in the project. Challenges were encountered during the process of instantiating the DSP blocks. Initially, difficulties were faced in correctly configuring the DSP by determining the appropriate parameters, as the available documentation occasionally lacked clarity. As a result, the synthesis tool failed to recognize the configuration. To overcome this obstacle, an examination of the source code of the open-source synthesis program was conducted to identify the cause of the error message, illustrated in Figure 6.6 in the appendix. After identifying the relevant parameters in the icestorm source file, successful instantiation of the DSPs was achieved. To verify the correct configuration, a program was developed to assess the accuracy of the ALU and indicate successful execution through a specific LED pattern. Further details regarding this testing methodology can be found in section 4.

3.2: Sequential multiplexers and forwarding units

Following the implementation of the DSP blocks, significant efforts were made to minimize the critical path of the processor. The rationale behind this approach stemmed from the understanding that increasing the clock cycle could lead to substantial improvements in the CPU's performance, thus overshadowing other minor enhancements. Synchronising every multiplexer and incorporating sequential elements into the forwarding unit, ALU, and branch predictor were pursued as strategies to achieve this objective. The underlying motivation was rooted in the fact that the CPU's maximum clock frequency is determined by the critical signal propagation delay,

and reducing it would greatly enhance the CPU's frequency potential. While this would resemble pipelining, the resulting CPU frequency would be significantly higher. However, this approach entailed a drawback in the form of a drastic decrease in the processor's CPI. The decision to synchronize all of the CPU's components hinged upon a trade-off analysis based on the typical path lengths, critical path length, and number of combinational components between existing pipeline registers. When the critical path length greatly exceeded the typical path length, it was reasonable to synchronize the components within that path to eliminate the extended critical delay. This was because the remaining paths would possess minimal propagation time, thereby enabling a substantial increase in clock rate. Conversely, if the critical path length was comparable to other path lengths in the circuit, synchronizing the combinational components of the CPU to remove the critical path would yield negligible benefits outweighed by a significantly worsened CPI.

After carefully considering the merits and drawbacks of this strategy, it was chosen to be pursued. The inability to accurately determine the average, mode, or median path length necessitated an exploration of potential improvements by implementing these changes and observing their impact on the CPU's performance. Regrettably, successful application of these modifications was not achieved, thus preventing a definitive assessment of the viability of this strategy. Despite synthesizing a design in which every multiplexer was synchronized and achieving a critical frequency of over 55MHz, surpassing initial expectations, the design itself was non-functional. Consequently, this result had to be disregarded in the overall report analysis. A simple fix was employed, however, where the frequency of the design was increased from 6MHz to 12MHz, as the base design was already capable of this. Upon careful reflection, I have identified the potential issues associated with the technique employed. The canonical 5-stage pipeline, encompassing fetch, decode, execute, memory access, and writeback stages (as depicted in Fig. 6.7, appendix), is common in most CPUs. This standardized pipeline ensures that, assuming proper processor design, there are relatively few inherent timing issues, provided pipeline hazards are infrequent. However, by synchronizing every combinational component in the circuit, including multiplexers, a scenario arises where each instruction executes through pipelines of varying lengths. This is contingent on the number of multiplexers the signal traverses and whether the forwarding unit is utilized for that specific instruction. It is this deviation from the uniform pipeline structure that likely led to the encountered challenges during the implementation of this modified CPU design, rendering unsuccessful program execution. I also experimented with synchronizing only one component at a time, yet encountered similar issues as previously mentioned.

In future attempts at such design changes, a recommended course of action would involve re-architecting the CPU's components. This restructuring would facilitate the determination of the number of components (i.e., pipeline depth) that each instruction's action must traverse. Subsequently, the pipeline could be further segmented using lengthy cascaded registers, similar to the current design. Although this approach would introduce increased complexity and potentially higher resource usage (due to the utilization of numerous registers, many of which exceed 150 bits in length), it would result in a deeper pipeline. Notably, modern CPUs, such as the A-series ARM processors (with pipeline depths ranging from 15 to 20 stages) and high-performance desktop processors (exceeding 30 stages), follow a similar approach to achieve enhanced performance.

3.3 Branch Prediction

I successfully implemented an enhanced branch prediction algorithm, surpassing the original implementation. The initial design relied on a 2-bit saturating counter (refer to Fig. 6.8 in the appendix) that lacked consideration for the specific branch instruction. Consequently, this predictor failed to effectively capture the distinct behaviour exhibited by different branch instructions within the program. However, given the relatively simple nature of

programs executed on this CPU, limited by the FPGA's resources that restrict the size of instruction and data memory, the original predictor proved reasonably effective.

To enhance branch prediction capabilities, I first implemented a local branch predictor mechanism. This involved constructing a table where the index corresponded to the last 5 bits of a branch instruction's address, and the entry comprised a 2-bit saturating counter that facilitated the prediction. The use of a 2-bit saturating counter offered hysteresis, oscillating between strongly taken, weakly taken, weakly not taken, and strongly not taken states. This ensured that a single incorrect prediction did not drastically alter the predictor's overall behaviour. Ideally, the predictor required at least two negative outcomes before changing its prediction, a characteristic exhibited in this implementation. Consequently, the predictor took longer to "warm-up" (i.e., populate the prediction table with the most probable states for each branch). However, from a broader perspective, this was not a concern, as programs likely to benefit significantly from branch prediction (thus improving performance) would inherently be long enough. The local branch history table (BHT) employed 5 index bits to index the table (BHT), which could potentially introduce aliasing issues in large programs. However, in this CPU design, this was not a problem since the expectation was to run relatively small programs. Utilizing 5 index bits resulted in a shallow table of only 32 words, striking a balance between resource usage and performance. Increasing the table size for small programs would not outweigh the drawbacks associated with heightened resource utilization, preventing the creation of a Pareto-optimal design. Similar reasoning applies to the utilization of 2-bit saturating counters, which were sufficiently large to be effective without unnecessary resource wastage.

In addition to the local predictor, I incorporated a global branch predictor to detect correlations among branch instructions. The global history register, implemented as a shift register, stored the last 5 branch outcomes, forming a 5-bit word used to index the pattern history table (PHT). The PHT stored predictions for each global history instance, enabling the establishment of correlations among branch instructions located at different addresses—an achievement unattainable by the local predictor alone. Combining the global history register with the lower 5 bits of the current program counter, using XOR operation, facilitated further correlation with the program's present state. A longer global history buffer would have been able to detect (and therefore predict) longer patters, but would require exponentially more resources as its history table would be larger. It would also be wasteful, as most of the history table would be unused. That is why more advanced prediction mechanisms do not resort to larger history tables, but find smarter ways to retain and use the branch history. I did not implement a branch target buffer due to time constraints, but this would have been a useful addition to the CPU, as memory access is often one of the slowest operations in a CPU. Detailed diagrams illustrating the structure of the global and local branch predictors can be found in Figures 6.9, and the Verilog code encompassing their implementation is provided in Figures 6.10.

However, the challenge emerged when determining the most suitable predictor between the two. To address this, I implemented a straightforward tournament prediction scheme, incorporating another 32-word depth table (indexed by the lower 5 bits of branch instructions) to determine the currently more accurate predictor—the local or global predictor. Admittedly, this aspect of my design is relatively weak and fails to leverage the presence of two distinct predictors. The current scheme merely selects a single predictor rather than effectively combining the information provided by both. Advanced designs such as gselect or gshare present more robust methods of predictor combination but come at the cost of significantly higher resource utilization. Consequently, such designs prove more applicable to larger, high-performance CPUs, particularly those implemented on ASICs, where resource usage is typically less constrained, as opposed to smaller experimental cores on FPGAs.

The implementation of the new branch predictors encountered minimal difficulties, with the main challenge lying in finding sufficiently complex programs to assess the predictors' performance. During the initial stages, I faced a

minor obstacle while researching specific branch predictor types to employ. Due to the simplicity of many existing predictors for modern processors and the proprietary nature of CPU intellectual property, locating comprehensive sources listing individual predictors along with their advantages and disadvantages proved arduous. Moreover, imprecise definitions of predictors in various sources hindered targeted searches by name. Consequently, I opted to conduct independent evaluations of each predictor based on their respective merits, attempting to ascertain their behaviour under different branch pattern scenarios. Although time-consuming, this approach significantly enhanced my understanding of branch predictor dynamics.

4. Results & Test Procedure

This section contains the results of the benchmark tests used to evaluate the performance of the improved CPU design. Some comments will be made about resource usage and power consumption to place the data into perspective, but a focus will be kept on performance metrics.

Performance (DSP):

Multiple test procedures were necessary to validate the implementation of the Digital Signal Processor (DSP). Firstly, a comprehensive test was conducted to ensure the proper functioning of the DSP and its ability to produce the required results. This test, presented in Figure 6.11 of the appendix, involved performing various basic addition and subtraction operations. Initially, the tests focused on separately evaluating the addition and subtraction capabilities of the lower and upper sections. Since the DSP performs 32-bit addition and subtraction operations, it necessitates the segregation of inputs into 16-bit words. Subsequently, the program assessed the addition of numbers involving carry operations from the lower 16-bits to the upper 16-bits. Similarly, subtraction tests were conducted, but with borrow operations instead of carry operations. The LED pulse waveform was utilized as an indicator of successful program execution, exhibiting distinct flashing patterns corresponding to the outcome of result equality tests. The functional verification of the DSP design was successful, affirming its robust functionality. To accurately measure the LED's on/off duration, a USB oscilloscope was employed.

The primary advantage of utilizing the DSP is the reduction in critical path delay, as depicted in Figure 6.12, which demonstrates a decrease from 74.14ns (13.49MHz) to 70.34ns (14.22MHz). However, this diminished critical delay would only lead to enhanced performance if the clock frequency could be increased accordingly. Unfortunately, the available high-frequency oscillator within the FPGA is limited to 6MHz, 12MHz, 24MHz, and 48MHz. Consequently, a slight increment in the maximum clock rate would not affect the design's performance, resulting in identical outcomes as depicted in the results table, Figure 3.1. Our team attempted to incorporate a Phase-Locked Loop (PLL) to gain more precise control over the clock frequency. Regrettably, the PLL's minimum frequency of 16MHz rendered it unsuitable for our design requirements. To evaluate performance, both the Bubblesort algorithm and a mathematical operations test program were executed, yielding the expected results. Although there were slight variations in the runtimes, they fell well within the expected margin of error. Multiple pulses were measured and averaged to enhance experimental accuracy. Moreover, a notable observation can be made regarding the resource utilization of the modified design incorporating the Digital Signal Processor (DSP) compared to the original design. As anticipated, the inclusion of 32-bit full adders, which require a considerable number of gates to perform their operations, contributes to higher resource consumption in the original design. This substantiates the advantage of employing DSPs, as they render the device more Pareto optimal by mitigating excessive resource usage.

Processor design	Original design	Modified design with DSP
Critical delay (ns)	74.14	70.34
Critical frequency (MHz)	13.49	14.22

Frequency of clock during runtime (MHz)	12	12
Bubblesort runtime (s)	2.09	2.08
ALU test (s)	0.645	0.639
Logic cell count	3128	3085

Figure 3.1: A table comparing the runtime of the original and modified designs for bubblesort

Performance (Updated branch predictor):

The testing process for the branch predictor is notably more intricate. Specifically, the evaluation of local and global history branch predictors necessitates distinct test scenarios due to their unique operational characteristics. For the local predictor, tests should exhibit consistent and predictable behaviour for specific branches to demonstrate its potential runtime improvement. Conversely, the global predictor requires tests that display correlation between branch instructions to showcase its enhancement capabilities.

To address these requirements, a program named "branch_test.c" was developed, encompassing and evaluating both of these predictor traits. Within this program, the "testGlobalBranchPredictor" function focuses on testing the global predictor by implementing a fixed branch history pattern limited to 5 bits. This pattern generates branches for every multiple of 2, 3, and 5 up to 10,000, establishing a consistent 5-bit branch history pattern that can be effectively predicted. On the other hand, the "testLocalBranchPredictor" function tests the local history predictor for each branching statement within the function. However, it also provides some evaluation of the global predictor, as there are conditional "if" statements that exhibit partial correlation. Overall, "branch_test.c" serves as a valuable tool for testing different branch prediction mechanisms, delivering improved runtime for the enhanced branch prediction approach. It is worth noting that the Bubblesort instructions amounted to 20,825,022, whereas the Branch_test instructions totalled 17,559,912.

Branch Predictor	Algorithm	Runtime	Logic Cell	CPI
Original	Bubblesort	2.115	3128	1.219
Local	Bubblesort	2.108	3245	1.215
Global	Bubblesort	2.023	3256	1.166
Tournament	Bubblesort	1.994	3602	1.149
Original	Branch_test	1.781	3445	1.217
Local	Branch_test	1.784	3602	1.219
Global	Branch_test	1.722	3615	1.177
Tournament	Branch_test	1.692	3961	1.156

Figure 3.2: Branch Prediction test results

Figure 3.2 reveals noteworthy insights regarding the prediction accuracy of both algorithms, indicating that the tournament predictor exhibits the highest accuracy. This is evident from its reduced program runtime, which suggests a lower occurrence of pipeline flushes caused by mispredictions. On the other hand, the local history predictor demonstrates insignificant gains in speed compared to the original predictor. This lack of significant improvement can be attributed to the similarities between the two predictors, with the only notable distinction being the presence of a saturating counter for each branch instruction in the local predictor. The likelihood of aliasing in the local history table's index, where different branch instructions share the same lower bits, is minimal due to the vast number of possible branch instruction addresses and the limited occurrence of shared lower 5 bits (considering the potential variability of up to 32 bits).

In contrast, the global predictor exhibits substantial differences from the original predictor, leading to improved outcomes. This improvement stems from both programs possessing global branching patterns, as previously described, which can be accurately predicted by the global predictor. The tournament predictor emerges as the superior choice by leveraging the local and global predictors based on their respective accuracies. By doing so, the number of cases where mispredictions occur is minimized, as it is unlikely for both predictors to be incorrect simultaneously. It is important to note that the modified prediction mechanisms, featuring 32-deep tables, impose a significant resource burden. Nonetheless, considering that branch prediction mechanisms play a vital role in maintaining high performance in pipelined processors, particularly for lengthy programs, sacrificing resources for increased performance is a justifiable trade-off. Additionally, the improvement in the branch predictor is reflected in the CPI (cycles per instruction) for both Bubblesort and Branch_test algorithms. Bubblesort showcases a decrease in CPI from 1.219 to 1.166, while Branch_test demonstrates a decline from 1.217 to 1.156. This improvement can be attributed to the enhanced pipeline performance, as mentioned earlier. These results validate the hypothesis presented in section 2, highlighting the distinct impact of branch prediction improvements on the CPU's overall performance, underscoring the significance of such enhancements.

5. Conclusions

The processor design improvements have demonstrated a direct enhancement in CPU performance. Notably, the implementation of the DSP has led to an improved maximum clock rate, increasing from 13.49MHz to 14.22MHz. Additionally, the CPI of the processor has shown improvements of 4.3% and 5.0% for the bubblesort and Branch_test programs, respectively. These advancements have propelled the design closer to being Pareto-optimal. However, they have necessitated increased resource utilization. Specifically, the adoption of relatively large prediction tables in the tournament predictor, which proved to be the most effective predictor, resulted in approximately 400 additional logic cells being utilized. This trade-off is deemed necessary as the tournament predictor represents a natural progression from the original branch predictor and significantly reduces the runtime for programs with extensive branching instructions.

Looking towards future improvements, there are several enhancements that can be made to the processor's design. Firstly, implementing all comparison operations in the ALU using the DSPs would be beneficial, as it would substantially reduce the critical delay path. Additionally, reconfiguring the layout of the CPU to accommodate more pipelining registers holds potential for significant performance gains, despite the impact on CPI. Considering the relatively low resource usage in relation to the FPGA's maximum capacity, the place-and-route tool can generate shorter routes between components, potentially reducing the critical path to under 15ns. Achieving this critical frequency of approximately 65MHz would require the utilization of a PLL, surpassing the limitations of the FPGA's high-frequency oscillator. However, future alterations to the branch predictor are not recommended. Advanced branch prediction mechanisms like gshare/gselect would incur significantly greater resource usage, rendering them unsuitable for simple CPU designs implemented on hobbyist-grade FPGAs. Additionally, it would be advantageous to employ BRAM (block RAM) for storing instruction and program memory, as the current storage in logic cells proves inefficient and resource-intensive within FPGAs.

Regarding team coordination, our group exhibited effective communication and prompt sharing of results. We provided valuable suggestions to one another when encountering obstacles. However, a weakness in our coordination lay in the absence of a centrally coordinated development plan. Instead, each member created individual plans. Establishing a central plan from the outset would have provided clarity regarding the required workload and ongoing component assignments. Nevertheless, overall, our team functioned cohesively and effectively.

6. Appendix

```
Total number of logic levels: 50
Total path delay: 74.14 ns (13.49 MHz)
```

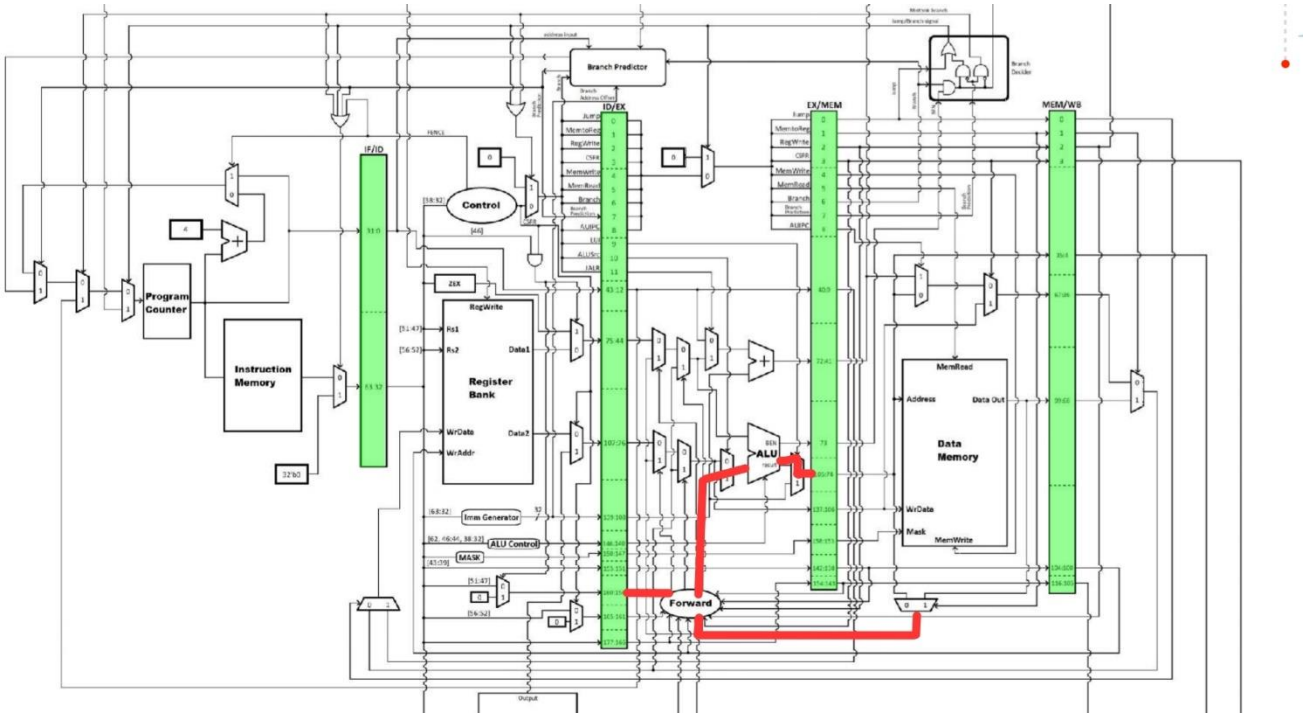

Fig. 6.1: Output from place-and-route tool for bubblesort critical path for the original processor

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Fig 6.2: Branch hazard in a pipeline – when not-taken is predicted. Source: Computer Architecture by Patterson and Hennessey

```
Info: Device utilisation:
Info:            ICESTORM_LC:  3128/ 5280     59%
Info:           ICESTORM_RAM:    20/   30     66%
Info:                  SB_IO:     8/   96      8%
Info:                  SB_GB:     5/    8     62%
Info:           ICESTORM_PLL:     0/    1      0%
Info:            SB_WARMBOOT:     0/    1      0%
Info:           ICESTORM_DSP:     0/    8      0%
Info:         ICESTORM_HFOSC:     1/    1    100%
Info:         ICESTORM_LFOSC:     0/    1      0%
Info:                 SB_I2C:     0/    2      0%
Info:                 SB_SPI:     0/    2      0%
Info:                 IO_I3C:     0/    2      0%
Info:            SB_LEDDA_IP:     0/    1      0%
Info:            SB_RGBA_DRV:     0/    1      0%
Info:         ICESTORM_SPRAM:     0/    4      0%
```

Fig. 6.3: Device utilisation during synthesis of CPU, without using DSP blocks

```
93      // DSP for ADDITION
94      SB_MAC16 add_dsp
95          (       // port interfaces
96          .A(A_in_lower),
97          .B(B_in_lower),
98          .C(C_in_lower),
99          .D(D_in_lower),
100         .O(add_dsp_out),
101         .CLK(clk),
102         .CE(one_reg),
103         .IRSTTOP(zero_reg),
104         .IRSTBOT(zero_reg),
105         .ORSTTOP(zero_reg),
106         .ORSTBOT(zero_reg),
107         .AHOLD(zero_reg),
108         .BHOLD(zero_reg),
109         .CHOLD(zero_reg),
110         .DHOLD(zero_reg),
111         .OHOLDTOP(zero_reg),
112         .OHOLDBOT(zero_reg),
113         .OLOADTOP(zero_reg),
114         .OLOADBOT(zero_reg),
115         .ADDSUBTOP(zero_reg),
116         .ADDSUBBOT(zero_reg),
117         .CO(add_CO),
118         .CI(zero_reg),
119         .ACCUMCI(),
120         .ACCUMCO(),
121         .SIGNEXTIN(),
122         .SIGNEXTOUT()
123         );
124         defparam add_dsp.NEG_TRIGGER = 1'b0;
125         defparam add_dsp.C_REG = 1'b0;
126         defparam add_dsp.A_REG = 1'b0;
127         defparam add_dsp.B_REG = 1'b0;
128         defparam add_dsp.D_REG = 1'b0;
129         defparam add_dsp.TOP_8x8_MULT_REG = 1'b0;
130         defparam add_dsp.BOT_8x8_MULT_REG = 1'b0;
131         defparam add_dsp.PIPELINE_16x16_MULT_REG1 = 1'b0;
132         defparam add_dsp.PIPELINE_16x16_MULT_REG2 = 1'b0;
133         defparam add_dsp.TOPOUTPUT_SELECT = 2'b00; // accum register output at O[31:16]
134         defparam add_dsp.TOPADDSUB_LOWERINPUT = 2'b00;
135         defparam add_dsp.TOPADDSUB_UPPERINPUT = 1'b1;
136         defparam add_dsp.TOPADDSUB_CARRYSELECT = 2'b10;
137         defparam add_dsp.BOTOUTPUT_SELECT = 2'b00; // accum regsiter output at O[15:0]
138         defparam add_dsp.BOTADDSUB_LOWERINPUT = 2'b00;
139         defparam add_dsp.BOTADDSUB_UPPERINPUT = 1'b1;
140         defparam add_dsp.BOTADDSUB_CARRYSELECT = 2'b00;
141         defparam add_dsp.MODE_8x8 = 1'b1;
142         defparam add_dsp.A_SIGNED = 1'b1;
143         defparam add_dsp.B_SIGNED = 1'b1;
```

Fig. 6.4: Explicit Verilog instantiation of DSP block for addition, with all of the parameters defined. Same was done for the subtraction DSP block.

```
Resolvable net names on path:
     1.491 ns ..  3.252 ns processor.ex_mem_out[140]
     4.537 ns ..  6.947 ns processor.forwarding_unit.MEM_RegWrite_SB_LUT4_I2_I3
     7.822 ns ..  9.583 ns processor.forwarding_unit.MEM_RegWrite_SB_LUT4_I2_O
    10.815 ns .. 13.570 ns processor.forwarding_unit.MEM_fwd2_SB_LUT4_O_I3
    14.854 ns .. 16.616 ns processor.mfwd2
    17.821 ns .. 19.583 ns processor.mem_fwd2_mux_out[0]
    20.814 ns .. 24.218 ns data_WrData[0]
    25.092 ns .. 28.947 ns processor.alu_mux_out[0]
    30.231 ns .. 31.993 ns processor.alu_main.ALUOut_SB_LUT4_O_17_I3_SB_LUT4_O_I0_SB_LUT4_O_I2_SB_CARRY_I1_CO_SB_CARRY_CO_29_I1
    34.615 ns .. 35.172 ns processor.alu_main.ALUOut_SB_LUT4_O_17_I3_SB_LUT4_O_I0_SB_LUT4_O_I2_SB_CARRY_I1_1_CO[8]
    37.397 ns .. 37.953 ns processor.alu_main.ALUOut_SB_LUT4_O_17_I3_SB_LUT4_O_I0_SB_LUT4_O_I2_SB_CARRY_I1_1_CO[16]
    40.178 ns .. 40.734 ns processor.alu_main.ALUOut_SB_LUT4_O_17_I3_SB_LUT4_O_I0_SB_LUT4_O_I2_SB_CARRY_I1_1_CO[24]
    42.681 ns .. 42.681 ns processor.alu_main.ALUOut_SB_LUT4_O_17_I3_SB_LUT4_O_I0_SB_LUT4_O_I2_SB_CARRY_I1_1_CO[31]
    42.959 ns .. 44.178 ns $nextpnr_ICESTORM_LC_1$I3
    45.052 ns .. 46.813 ns processor.alu_main.Branch_Enable_SB_LUT4_O_I0_SB_LUT4_O_I0_SB_LUT4_O_I1_SB_LUT4_O_I0
    48.098 ns .. 51.502 ns processor.alu_main.Branch_Enable_SB_LUT4_O_I0_SB_LUT4_O_I0_SB_LUT4_O_I1
    52.734 ns .. 54.495 ns processor.alu_main.ALUOut_SB_LUT4_O_12_I3_SB_LUT4_O_I2_SB_LUT4_O_I3
    55.369 ns .. 57.131 ns processor.alu_main.ALUOut_SB_LUT4_O_12_I3_SB_LUT4_O_I2
    58.336 ns .. 60.098 ns processor.alu_main.ALUOut_SB_LUT4_O_12_I3
    61.329 ns .. 63.091 ns processor.alu_result[0]
    64.323 ns .. 66.084 ns processor.alu_main.Branch_Enable_SB_LUT4_O_I1_SB_LUT4_O_I2_SB_LUT4_O_I3
    67.316 ns .. 69.077 ns processor.alu_main.Branch_Enable_SB_LUT4_O_I1_SB_LUT4_O_I2
    70.283 ns .. 73.276 ns processor.alu_main.Branch_Enable_SB_LUT4_O_I1
                  lcout -> processor.ex_mem_out[73]
```

Fig. 6.5: Bubblesort critical path

```
Warning: detected unknown/unsupported DSP config, defaulting to 16x16 MUL.
Warning: detected unknown/unsupported DSP config, defaulting to 16x16 MUL.
```

```
} else if (mode_8x8 && botuprin && (botlwrin == 0) && (topcarry == 2)) {
        basename = "SB_MAC16_ADS_U_32P32";
} else if (mode_8x8 && botuprin && (botlwrin == 1)) {
        basename = "SB_MAC16_MAS_U_8X8";
} else if (!mode_8x8 && botuprin && (botlwrin == 2)) {
        basename = "SB_MAC16_MAS_U_16X16";
} else {
        fprintf(stderr, "Warning: detected unknown/unsupported DSP config, defaulting to 16x16 MUL.\n");
}
```

Fig. 6.6: Error message when DSP was configured wrong

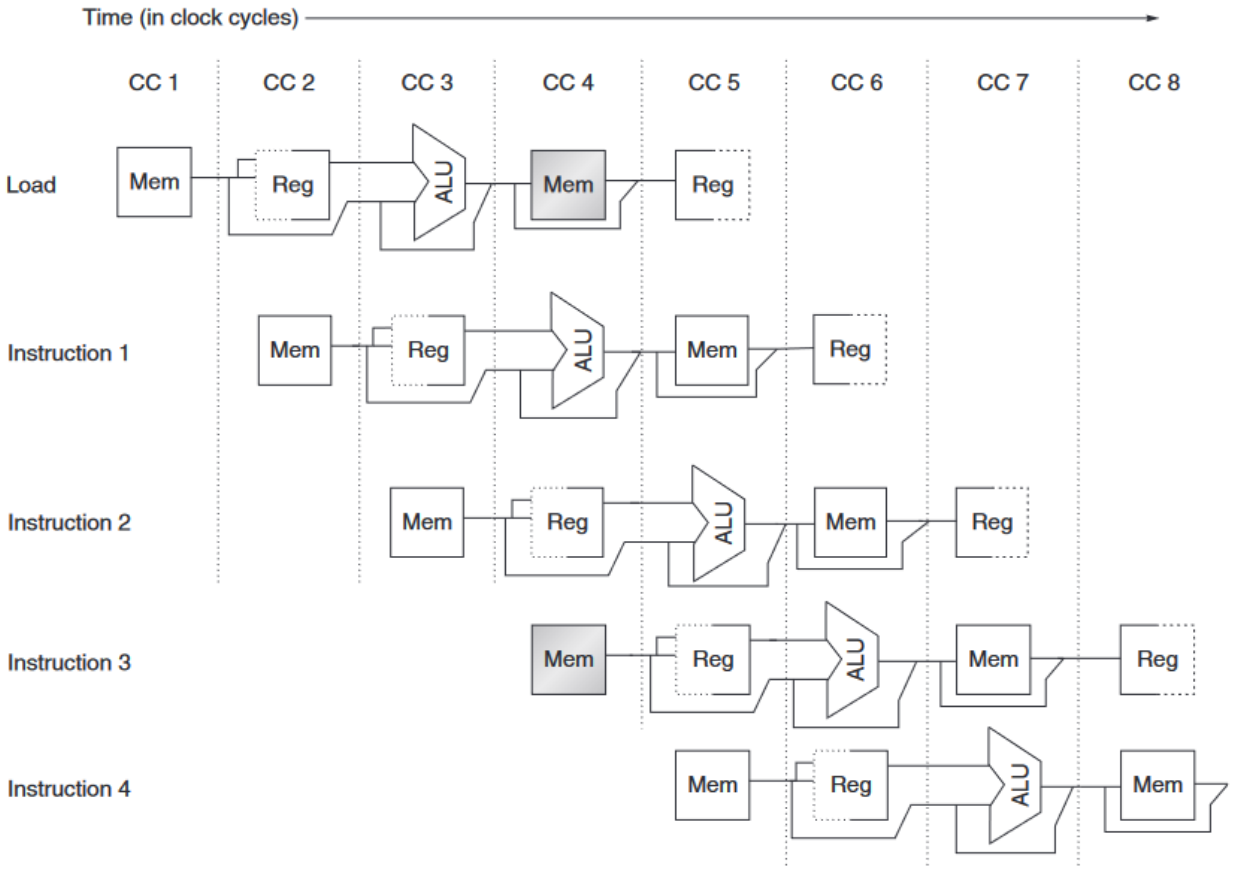

Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

Fig. 6.7: 5-stage pipeline, Source: Computer Architecture by Patterson and Hennessey

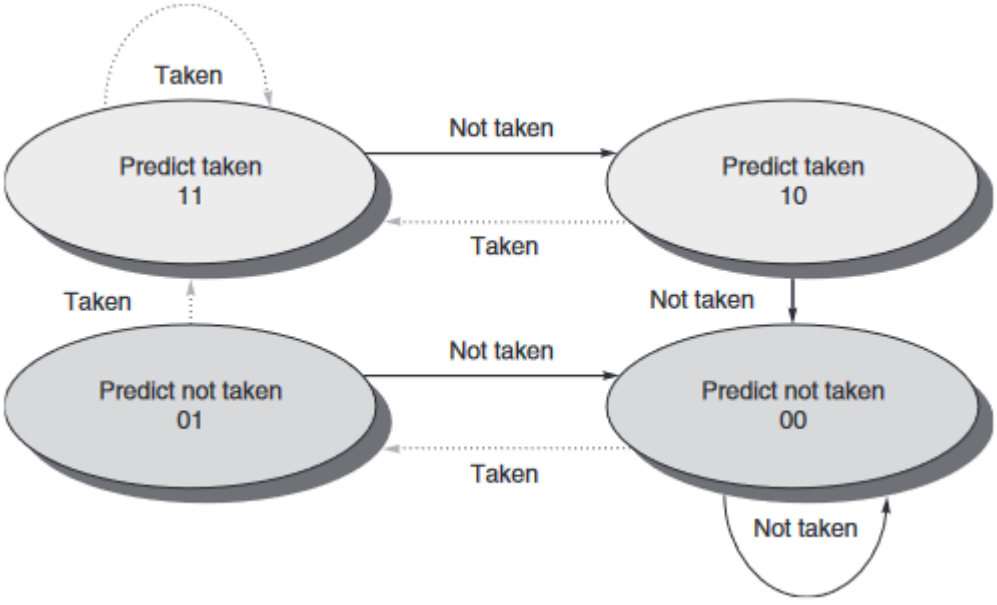

Fig. 6.8: 2-bit saturating counter

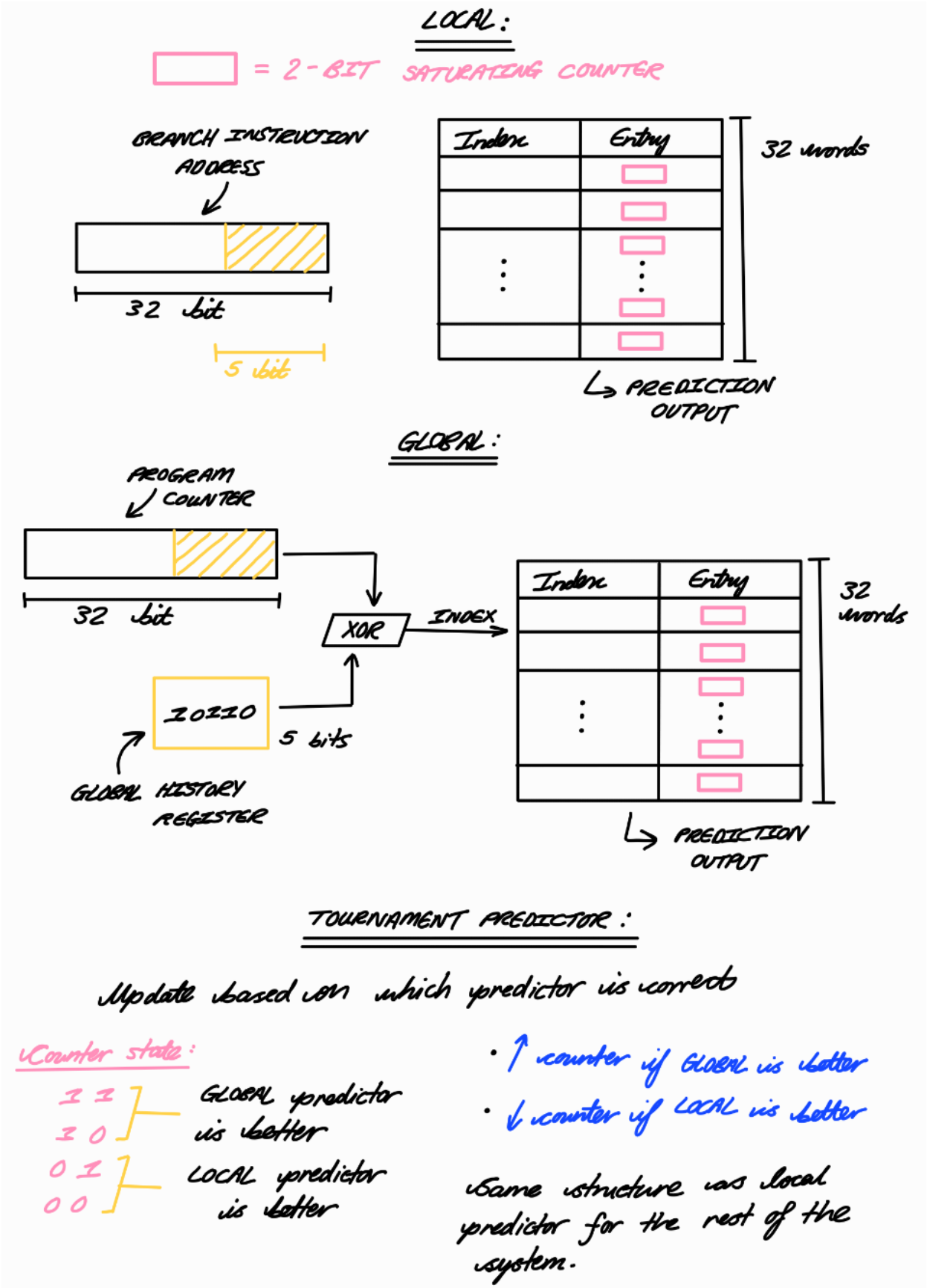

Figure 6.9: Local, Global and tournament predictor structures

```
    reg     branch_mem_sig_reg;
    // branch history table, each entry is a 2-bit saturating counter
    reg [1:0] bht [31:0];
    // global branch history table, each entry is a 2-bit saturating counter
    reg [1:0] gbht [31:0];
    // tournament history table, each entry is a 2-bit saturating counter
    reg [1:0] tournament_ht [31:0];
    // global history register, only 5 bits as gbht would be too large otherwise
    reg [4:0] ghr;
    wire [4:0] bht_index;
    wire [4:0] gbht_index;

    initial begin
        branch_mem_sig_reg = 1'b0;
    end

    assign bht_index = in_addr[4:0];
    assign gbht_index = in_addr[4:0] ^ ghr;
    assign branch_addr = in_addr + offset;
    assign local_prediction = bht[bht_index][1];
    assign global_prediction = gbht[gbht_index][1];

    always @(negedge clk) begin
        branch_mem_sig_reg <= branch_mem_sig;
    end

    always @(posedge clk) begin
        if (branch_mem_sig_reg) begin
            // update 2-bit saturating counter inside
            // each entry of bht or gbht based on actual branch decision
            if (actual_branch_decision == 1) begin
                if (bht[bht_index] < 3) begin
                    bht[bht_index] <= bht[bht_index] + 1;
                end

                if (gbht[gbht_index] < 3) begin
                    gbht[gbht_index] <= gbht[gbht_index] + 1;
                end
            end

            else begin
                if (bht[bht_index] > 0) begin
                    bht[bht_index] <= bht[bht_index] - 1;
                end

                if (gbht[gbht_index] > 0) begin
                    gbht[gbht_index] <= gbht[gbht_index] - 1;
                end
            end
        end
        ghr <= {ghr[3:0], actual_branch_decision};
    end

    // tournament predictor
    always @(posedge clk) begin
        if (branch_mem_sig_reg) begin
            if (actual_branch_decision == 1) begin
                // 10 & 11 corresponds to local prediction being taken
                // 00 & 01 corresponds to global prediction being taken
                // Can use bht_index as index for tournament_ht as well
                if (local_prediction == 1 && global_prediction == 0) begin
                    if (tournament_ht[bht_index] < 3) begin
                        tournament_ht[bht_index] <= tournament_ht[bht_index] + 1;
                    end
                end else if (local_prediction == 0 && global_prediction == 1) begin
                    if (tournament_ht[bht_index] > 0) begin
                        tournament_ht[bht_index] <= tournament_ht[bht_index] - 1;
                    end
                end
            end

            else if (actual_branch_decision == 0) begin
                if (local_prediction == 1 && global_prediction == 0) begin
                    if (tournament_ht[bht_index] > 0) begin
                        tournament_ht[bht_index] <= tournament_ht[bht_index] - 1;
                    end
                end else if (local_prediction == 0 && global_prediction == 1) begin
                    if (tournament_ht[bht_index] < 3) begin
                        tournament_ht[bht_index] <= tournament_ht[bht_index] + 1;
                    end
                end
            end
        end
    end
    assign prediction = tournament_ht[bht_index][1] & branch_decode_sig;
endmodule
```

Figure 6.10: Local and global branch predictors with a tournament predictor that combines them

```c
void flash_fast() {
    volatile unsigned int *gDebugLedsMemoryMappedRegister = (unsigned int *)0x2000;
    *gDebugLedsMemoryMappedRegister = 0xFF;
    for (int k = 0; k < 6; k++) {
        //Null statement to waste time
        *gDebugLedsMemoryMappedRegister = ~(*gDebugLedsMemoryMappedRegister);
        for (int j = 0; j < 10000; j++) {
        //Null statement to waste time
        ;
        }
    }
}

void flash_slow() {
    volatile unsigned int *gDebugLedsMemoryMappedRegister = (unsigned int *)0x2000;
    *gDebugLedsMemoryMappedRegister = 0xFF;
    for (int j = 0; j < 100000; j++) {
        ;
    }
    *gDebugLedsMemoryMappedRegister = ~(*gDebugLedsMemoryMappedRegister);
    for (int l = 0; l < 100000; l++) {
        ;
    }
}

int main() {
    int a = 1;
    int b = 3;
    int sum = 0;

    sum = a + b;
    if (sum == 4) {
        flash_slow();
    } else {
        flash_fast();
    }

    a = 717225898;
    b = 717225898;

    sum = a + b;
    if (sum == 1434451796) {
        flash_slow();
    } else {
        flash_fast();
    }

    a = 5;
    b = 1;
    sum = a - b;
    if (sum == 4) {
        flash_slow();
    } else {
        flash_fast();
    }

    a = 715827882;
    b = 357913941;
    sum = a - b;
    if (sum == 357913941) {
        flash_slow();
    } else {
        flash_fast();
    }

    return 0;
}
```

Figure 6.11: modified_alu_test.c